Detecting Data and Schema Changes in Scientific Documents

Nabil Adam, Igg Adiwijaya
CIMIC - Rutgers University
180 Univ. Ave, Newark, NJ 07102
adam@adam.rutgers.edu,
gusadi@cimic.rutgers.edu

Terence Critchlow, Ron Musick
Lawrence Livermore National Laboratory
7000 East Ave. Livermore, CA 94550
critchlow@llnl.gov,
rmusick@llnl.gov

Abstract

Data stored in a data warehouse must be kept consistent and up-to-date with respect to the underlying information sources. By providing the capability to identify, categorize and detect changes in these sources, only the modified data needs to be transfered and entered into the warehouse. Another alternative, periodically reloading from scratch, is obviously inefficient. When the schema of an information source changes, all components that interact with, or make use of, data originating from that source must be updated to conform. The change detection problem is the problem of detecting data and schema changes by comparing two versions of the same semi-structured document. In this paper, we present an approach to detecting data and schema changes for scientific documents. Scientific data is of particular interest because it is normally stored as semi-structured document, and suffers frequent schema updates. This paper demonstrates the use of graph to represent scientific documents in particular, and semi-structured documents in general as well as their schema. It also demonstrates an approach to efficiently detect data and schema changes by merging the detection with parsing the document.

1. Introduction

Detecting change is an important task for many applications, in particular data warehousing. A data warehouse integrates data from heterogeneous, autonomous data sources into a consistent, central repository. Since the warehouse needs to be kept consistent and up-to-date, changes to the underlying sources must be periodically extracted and propagated to the warehouse. While this could be done by regularly refreshing the entire warehouse, a better alternative is to detect and propagate only the changes since this requires less computer resources and is within the required time constraints – typically one business night.

Before data can be loaded, the source schema must be obtained and incorporated into the components supporting the warehouse, such as the wrapper and mediator [6]. When the schema evolves, these components must be modified as well. Currently, reflecting source schema changes in the warehouse requires obtaining the updated schema from the source and manually modifying the warehouse components to conform to it. While a manual approach to detecting schema changes may be acceptable in certain environments, it is too cumbersome and inefficient in situations where schema changes are frequent, such as scientific environments. An approach to automatically detecting schema changes and semi-automatically modifying the warehouse components accordingly is needed. The more we can automate this process, the more useful it will be. Furthermore, such an approach would make adding new sources to the warehouse easier as well. However, since the schema is usually not explicitly specified in a free-form document, there are significant technical challenges to overcome if we are to reach any level of automation.

In this paper, we present our approach to detecting both data and schema changes in scientific data sources, with a focus on genomic sources. In this environment, data is usually provided as semi-structured documents adhering to a well-defined, but representationally complex, schema. Therefore, we consider the problem of detecting change only within the context of semi-structured documents. To detect these changes, we compare two versions of the same document. The comparison utilizes any characteristics, rules and relationships existing in the documents, as described by the schema. We use a graph representation to describe the schema, and view documents as instances of this graph. Since a document contains data reflecting part of the schema (for example through tags), we use it as an indication of schema changes.

The rest of this paper is organized as follows. We begin by briefly summarizing some of the traditional change detection algorithms. In Section 3, we describe the characteristics of scientific documents, with an emphasis on genomic databases, and discuss how we formally represent them.

Section 4, presents our approach to data change detection, including a description of the set of possible data changes. Section 5 presents the set of possible schema changes and our approach to detecting them. We conclude with an outline of future steps in our on-going project in Section 7.

2. Related Work

There have been several papers which detect data changes by comparing two versions of the same document. Initial work in this area focussed on changes in unstructured documents. Myers et al. [12, 18] detect changes between strings using the longest common subsequence (LCS) [11] algorithm, and consider only insertion and deletion operations. Wagner et al. [16] use insertion, deletion and update operations to find the best sequence of operations that can transform one string into another.

More recently, there have been several approaches to detecting data changes in semi-structured documents. These approaches differ primarily in how they view the underlying documents. Ball et al. [9, 10] view semi-structured documents as containing sequence of sentences and "sentence-breaking" markups. A sentence is a non-recursive set of words and non-sentence-breaking markups. Sentence-breaking markups separate sentences from each other and from collections of sentences. In comparing two documents, the LCS algorithm is used to determine the total number of matched words and markups within these sentences compared to the total length of the two sentences. With this approach, a sentence may need to be compared with all sentences in the other document. Even though changes to data can be detected, changes to the schema cannot be.

Alternatively, Chawathe et al. [7] and Zhang et al. [15, 17, 19] view semi-structured documents as trees. Thus, the problem of change detection has been transformed to the problem of finding differences between trees. For any pair of leaf nodes, they either match or not, as determined by LCS. Two internal nodes strictly match if all their children match, and partially match if at least some children match. To detect data changes, the two documents are first converted into their corresponding trees, then the matching nodes are identified. Once this has been done, all of the sequences of operations (insert, delete, update, move) that convert the old tree to the new one are identified, usually resulting in several options. The sequence that best represents the transformation is the one with the lowest total cost, based on the cost assigned to each operation. This approach may be expensive since each node needs to be compared to all nodes in the other tree, and may match many of them, resulting in a large set of valid transformations.

In addition, given the nature of scientific documents as discussed in the next section, we need to be able to represent not only the data within semi-structured documents, but also the schema used to generate the documents. The representation should allow one to perform the task of detecting changes to data and schema. For the previous tree structure, even though the data within the document and their relationships can be represented in the tree, it cannot express all the possible schema and rules used for the document. It can only partially express that part of the schema that the document is using. For example, the PDB document of Figure 1 does not have $FRAGMENT$, which is an optional data item following $COMPND$. Using the tree structure, this type of schema information is not represented.

Chawathe et al. [1, 4, 5] and Nestorov et al. [14, 13] use edge labeled graphs to represent semi-structured documents. Although the use of edge labeled graphs can help in querying data within semi-structured documents, determining and assigning the labels to the edges requires significant manual effort. One needs to manually evaluate the content of a semi-structured document (e.g. an HTML page containing restaurant reviews as used in [5]) prior to constructing the edge labeled graph. In the restaurant example used in [5], the word "restaurant" does not even exist in some of the paragraphs within the HTML document. Instead, it has been manually decided that a marker, (<P>) in the HTML document would always indicate the beginning of a restaurant. Thus, a change to the structure of the HTML document would require manual re-evaluation of the document prior to necessary modification of the edge labeled graph representation. Furthermore, other important properties, such as object ordering, cannot be expressed using the edge labeled graph representation. For example, the author may have re-ordered the restaurants, from top to bottom based on his/her preferences. Alternatively, the author may replace <P> with <OL> (ordered list elements) to denote the ordering of data items is significant. Using the current labeled graph representation, such properties would be lost. In scientific documents, properties such as ordering may be very significant.

Our approach to detecting change does not require transforming one representation to match another, or finding the sequence of operations with minimum cost. Rather, we perform the tasks of node matching and comparison during the parsing of the documents. Furthermore, none of the approaches previously mentioned detect changes to the underlying document schema. While our examples focus on genomic scientific-documents, our approach is also applicable to other domains, such as HTML and XML documents, with some level of modification. The reasons for our focus on scientific documents are: (1) there are frequent schema changes in scientific environment, which present a more challenging problem, and (2) we feel that the schema used within scientific documents resides somewhere between HTML and XML. While in HTML the grammar can be very relaxed, for example, a markup can appear almost

anywhere within the body of an HTML document. In XML an extremely rigid DTD may be defined. Portions of a schema for scientific documents can be very relaxed while others are very rigid. While not all semi-structured documents contain a restrictive section, our algorithm makes use of this information, where it exists, to identify a greater number of changes with less effort, as outlined in Section 5.3. Scientific documents represent a good cross-section of semi-structured documents because of the interesting mix of flexible and restrictive areas.

3. Semi-structured scientific documents

As described in Section 2, there are several ways one can view and model a semi-structured document. Our view is based on experience with scientific documents in general, and genomic documents in particular. To provide a consistent and concrete framework for presenting our representation, we use a single example, the Protein Data Bank (PDB) [3] data source shown in Figure 1. We present the characteristics and rules of the schema below.

```
HEADER    DEOXYRIBONUCLEIC ACID                    26-JUN-96   302D
TITLE     META-HYDROXY ANALOGUE OF HOECHST 33258 ('HYDROXYL IN'
TITLE    2 CONFORMATION) BOUND TO D(CGCGAATTCGCG)2
COMPND    MOL_ID: 1;
COMPND   2 MOLECULE: DNA
COMPND   3 (5'-D(*CP*GP*CP*GP*AP*AP*TP*TP*CP*GP*CP*G)-3');
COMPND   4 CHAIN: A, B;
COMPND   5 ENGINEERED: YES
SOURCE    MOL_ID: 1;
SOURCE   2 SYNTHETIC: YES
KEYWDS    B-DNA, DOUBLE HELIX, COMPLEXED WITH DRUG, DEOXYRIBONUCLEIC
KEYWDS   2 ACID
EXPDTA    X-RAY DIFFRACTION
AUTHOR    G.R.CLARK,C.J.SQUIRE,E.J.GRAY,W.LEUPIN,S.NEIDLE
REVDAT   1   12-FEB-97 302D    0
JRNL        AUTH   G.R.CLARK,C.J.SQUIRE,E.J.GRAY,W.LEUPIN,S.NEIDLE
JRNL        TITL   DESIGNER DNA-BINDING DRUGS: THE CRYSTAL STRUCTURE
JRNL        TITL 2 OF A META-HYDROXY ANALOGUE OF HOECHST 33258 BOUND
JRNL        TITL 3 TO D(CGCGAATTCGCG)2
JRNL        REF    NUCLEIC ACIDS RES.            V.  24  4882 1996
JRNL        REFN   ASTM NARHAD  UK ISSN 0305-1048                0389
REMARK   1
REMARK   1 REFERENCE 1
REMARK   1  AUTH   S.E.EBRAHIMI,M.C.BIBBY,K.R.FOX,K.T.DOUGLAS
REMARK   1  TITL   SYNTHESIS, DNA BINDING, FOOTPRINTING AND IN VITRO
REMARK   1  TITL 2 ANTITUMOUR STUDIES OF A META-HYDROXY ANALOGUE OF
REMARK   1  TITL 3 HOECHST 33258
REMARK   1  REF    ANTI-CANCER DRUG DES.         V.  10   463 1995
```

Figure 1. Sample of a PDB document

A schema for semi-structured scientific documents, s, consist of a set of data objects, O, and a set of constraints, C, between the data objects. A data object, o, is comprised of an identifier, $ident(o)$, and a value, $val(o)$. Values are optional. For example, in Figure 1, data object o_i with $ident(o_i) = SOURCE$ has no value (i.e. $val(o_i) = null$) because there is no data associated with it, while object o_j, with $ident(o_j) = SYNTHETIC$, has a value of *YES*. In some cases, ordering among objects is significant, in others it is not. For example, the compound description ($COMPND$) must come after the title and before the source. However, it doesn't matter if its molecule type comes before the chain-identifier information or after it. Each schema object is contained in exactly one of the following sets:

1. *Mandatory objects*. These objects must exist in a document. For example, in a valid PDB document there must be a data object o_i with $ident(o_i) = HEADER$.

2. *Optional objects*. These objects may or may not exist in a valid document. For example, o_i where $ident(o_i) = REMARK4$ does not have to exist for a PDB document to be valid.

3. *Conditional objects*. Each of these objects can only exist if another object exists. For example, in a PDB document, a molecule type (*SYNTHETIC*) exists if and only if there is a corresponding compound description (*SOURCE*). A conditional object can be either mandatory or optional. For a conditional object to be mandatory, the object it conditionally exists upon must be a mandatory object.

We feel that a model for scientific documents should posses two properties; the capability to represent both the data within the documents and the schema to generate the documents, and should support change detection for both data and schema. This does not mean that we ignore other characteristics, such as support for querying data within the documents, as in [5]. As briefly discussed later, our representation could be converted into other representations, including labeled-edge graphs or trees.

In order to have the previous properties, we view a schema of scientific documents as simply a directed graph, called *schema graph*. A schema graph is a natural and direct translation of the grammar for scientific documents into a graph. Consequently, we view a scientific document as an instance of its schema graph. Nodes in the graph consists of a label and, possibly, a value. When necessary, the schema graph can be converted into edge labeled graph by simply removing the label from every node and place it on the corresponding edge. Formally, we define a schema graph *S* as consisting of:

- Nodes, n.
  Nodes correspond to the schema objects previously defined. There are three different types of nodes: *regular*, *optional* and *stopping* nodes.

  – Regular nodes, n^r, must have an associated identifier. If an identifier is not explicitly specified in a document, a generated one will be supplied. Values are optional. For example, the identifier *SOURCE* does not have a value while its children, such as *SYNTHETIC*, do. We use n_i^r and

$ident(n_i^r)$ to denote a regular node i and its identifier respectively. A regular node is represented by a circle on the schema graph.

  - Optional nodes, n^o, do not have identifiers or values. An optional node must have at least two children. For a given document, exactly one of the children nodes of an optional node must be selected. On a schema graph, an optional node is represented by a triangle.
  - Stopping nodes, n^F, do not have identifiers or values. These nodes function as a no-op and are needed as a termination point for repeatable nodes. Usually, one of the children of an optional node is a stopping node. A stopping node is represented by a circle with the letter *F* in it on the schema graph.

- Directed edges, e.
  A directed edge connects two nodes, n_i and n_j, where the edge begins at n_i, the parent, and ends at n_j, the child, and $n_i, n_j \in \{n^r, n^o, n^F\}$. For any n_i whose immediate parent n_j is not an optional node, whenever n_j exists in a document, n_i must also exist. Such a relationship is denoted by $n_i \rightarrow n_j$. There are two types of directed edges; *ordered* and *unordered* edges.

  - Ordered directed-edges, e^o, indicate that the ordering of the children from left to right is significant and must be preserved. These edges are represented by a solid line on the schema graph.
  - Unordered directed-edges, e^u, indicate that the ordering of the children from left to right is not critical. These edges are represented by dashed lines on the schema graph.

Thus a schema graph is a tuple $\langle N, E \rangle$, where N is a set of nodes and E is a set of directed edges. Figure 2 depicts an example of a simple schema with the following properties and rules. $n_1 - n_{14}$ are regular nodes and $n_1, n_2, n_3, n_4, n_5, n_9, n_{10}, n_{13}, n_{14}$ are required. n_2 must appear before n_3 and, if present, (n_6, n_{12}) and n_7 must appear between n_2 and n_3. (n_9, n_{13}, n_{14}) and n_{10} may appear in any order after n_3. (n_6, n_{12}) and n_{11} can exist multiple times with different values.

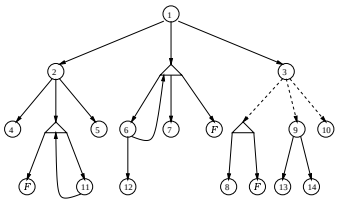

Figure 2. An example of a graph representation of a simple schema

We represent a semi-structured document by mapping it to an instance of the schema graph. The resulting *document graph* represents a subset of the schema graph, since optional rules and properties specified in the schema may not be directly reflected in the document. Figure 3 (a) shows a document graph conforming to the schema depicted by Figure 2. Because a document graph unrolls the loops in a schema graph, it is represented as a tree instead of a general graph structure. Figure 3 (b) depicts the tree schema, obtained by mapping the schema graph in Figure 2 to the document graph in Figure 3 (a). As can be seen from Figure 3 (a), a document graph, denoted by t_d, consists of the following.

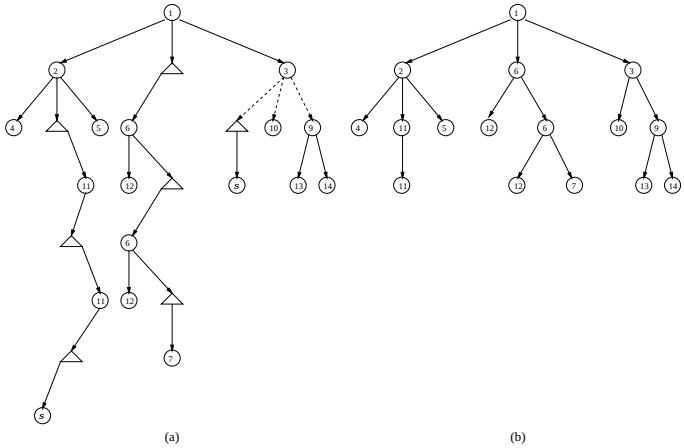

Figure 3. The subset graph and tree for a document

- Nodes, n. $n \in \{n^r, n^o, n^F\}$.
- Edges, e. $e \in \{e^o, e^u\}$.

4. Changes to data

To detect data changes, we use the schema to guide the comparison between different versions of the document. This requires extending the current parser to read the original document and store it internally. This effectively combines the schema graph and the original document graph to produce a "value-added" schema graph. Then the new document is read using the value-added schema, with the data values being compared during the parsing. This allows us to detect and evaluate changes while the document is being

loaded. Before we discuss detecting data changes in Section 4.2, we present our approach to producing the value-added schema and describe the types of changes we consider. Since a document graph is an instance of a schema graph, mapping the document graph to the schema graph is straightforward:

1. Parse the original document using the schema graph. For every object within the document, mark the corresponding node in the schema graph. For every value in the document, copy it to the corresponding node.

2. Extend optional nodes as necessary to handle loops.

To illustrate, suppose we would like to obtain the value added schema, $s^i_{d_j}$, where s^i is the schema given in Figure 2 and t_{d_j} is the document graph shown in Figure 3 (a). Figure 4 presents $s^i_{d_j}$ diagrammatically, with shaded circles and solid lines corresponding to the document's nodes and edges.

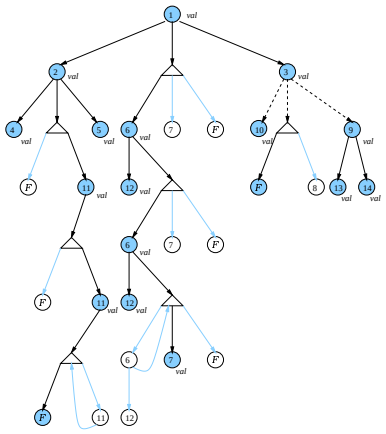

Figure 4. A schema having the knowledge of the content of a document

4.1. Types of changes to data

We define an output of a data change to be a tuple $\langle t, n, p, v, Sib\rangle$, where t, n, p, and v are the type of data change, corresponding node identifier, parent node of n, and value of n, respectively. Sib is a set of nodes, and their values, that that uniquely identify n's position in the value-added schema graph. This is necessary to differentiate repeatable nodes. Both v and Sib are optional.

Next, we briefly describe the types of data changes we consider in our approach and the output each generates:

- **Update Value**: $upd(val(n_i))$ occurs when an update is made to the value of a node n_i. This change requires the same node to appear in both versions, and $n_i \in \{n^r\}$.
  Output for $upd(val(n_i))$ is $\langle update, n_i, p_{n_i}, val(n_i), Sib\rangle$ and $val(n_i)$ is required.

- **Insert node**: $ins(n_i)$ occurs when a node n_i does not exist in the original document but exists in the newer one.
  Output for $ins(n_i)$ is $\langle insert, n_i, p_{n_i}, val(n_i), Sib\rangle$ and $val(n_i)$ is required.

- **Delete node**: $del(n_i)$ occurs when a child of an optional node n_i exists in the original document but not in the newer one. This also indicates that an n_F has been selected to replace n_i in the newer version. If n_i is a mandatory node, this change would indicate a schema change as discussed in the following section.
  Output for $del(n_i)$ is $\langle delete, n_i, p_{n_i}, Sib\rangle$.

- **Reorder nodes**: $reorder(n_i)$ occurs when unordered nodes have been reordered. If ordered nodes changed their position, this change would represent a schema change. In detecting $reorder(n_i)$, siblings of n_i need to be evaluated.
  Output for $reorder(n_i)$ is $\langle reorder, n_i, p_{n_i}, Sib\rangle$ and Sib is required to show the new ordering.

- **Move node**: $move(n_i)$ occurs when a node n_i is relocated upward or downward on the graph. This is only possible when the node is a child of an optional node.
  Output for $move(n_i)$ is $\langle move, n_i, p_{n_i}, Sib\rangle$.

4.2. Detecting changes to data

Before discussing our approach to detecting changes, we first clarify what it means for two nodes to match. Two nodes match if:

1. the identifiers match, and

2. the values are *relatively* equal, where this implies they match under the LCS algorithm.

To detect data changes within a document, the value-added schema can be used to parse the document and return the updates. The following pseudo-code provides a high level overview of the algorithm we use to detect data changes. The remainder of this section describes, in detail, the various cases handled by this algorithm.

$S \leftarrow s^i_{d^k_j}$ /* value added schema = schema + old doc. */
$D \leftarrow d^l_j$ /* new document */
While traversing S, parse D using S

for each $o_i \in D$
find o_j, the corresponding node in S
if o_j is an ordered node **then**
find $upd(o_i)$ against the corresponding marked o_j
if o_i is an unordered node **then**
find $upd(o_i), reorder(o_i)$ against o_j and its sibling
if o_i is a non-repeatable, optional node **then**
find $ins(), del()$ or $upd()$ of child node of
o_i against the child nodes of o_j
if o_i is a repeatable, optional node **then**
find $del(), ins(), move()$ and/or $upd()$ of subtree
sub-rooted at o_i against the corresponding subtree
sub-rooted at o_j

Ordered nodes
Detecting changes in ordered nodes is straightforward since they must occur in a specified sequence. Let d_j^k denote the kth version of document j. Given s^i, d_j^k, d_j^l ($k < l$), to detect changes to ordered nodes in d_j^l:

Traverse $s^i_{d_j^k}$ while parsing d_j^l
For every ordered node $n_v \in s^i_{d_j^k}$ and its corresponding
node $n_w \in d_j^l$
If val(n_v) does not match val(n_w),
then return $upd(val(n_w))$

Unordered nodes
Given s^i, d_j^k, d_j^l ($k < l$), to detect changes to unordered nodes in d_j^l:

Traverse $s^i_{d_j^k}$ while parsing d_j^l.
Foreach visited, mandatory unordered node n_v on $s^i_{d_j^k}$,
Fetch the corresponding object, n_w, in d_j^l.
Fetch n_{wc}, the children of n_w.
Compare each of n_v's children with n_{wc} using only
the identifier to find the corresponding node.
Record the order of the node and the object.
Compare the values of the matching node.
If the orders are different, return $reorder(n_w)$.
If the values of matching nodes are different,
add an $upd(n_w)$.

Non-repeatable, optional nodes
Given a schema graph s^i, a non-repeatable optional node n_k is a node such that n_k's ancestor $n_l \in n^o$ (i.e. it is optional) and $\forall n_m \mid n_m$ is a descendent of n_l and $\neg\exists$ an edge from n_m to n_l (i.e. it is not in a loop). Given $s_i, d_j^k, d_j^l (k < l)$, to detect changes to non-repeatable, optional nodes:

Traverse $s^i_{d_j^k}$ and parse d_j^l.
For each visited, non-repeatable, optional node n_v
Identify the corresponding optional node, n_w in d_j^l.
If $ident(n_v) = ident(n_w)$ and $val(n_v) \neq val(n_w)$ then
return $upd(n_w)$.
If $ident(n_v) \neq ident(n_w)$ then
return $del(n_v), ins(n_w)$.
If $n_w = null$ then return $del(n_v)$.
If $n_v = null$ and $n_w \neq null$ then return $ins(n_w)$.

Repeatable, optional nodes
Given a schema graph s^i, a non-repeatable optional node, n_k, is a node such that n_k's ancestor $n_l \in n^o$ and $\exists n_m \mid n_m$ is a descendent of n_l and $\exists$ an edge from n_m to n_l. For example, Figure 5 (a) depicts an s^i showing an optional node having a repeatable child n_6. Figure 5 (b) depicts a portion of $s^i_{d_j^k}$ with repeated nodes, and Figure 5 (c) depicts the tree representation of d_j^l. To detect changes to repeatable nodes, we use a *bucket* as temporary storage during the change detection process: b_j^l is associated with d_j^l. In addition, we also use a marker to mark the point of the last comparison within a set of repeatable nodes in $s^i_{d_j^k}$.

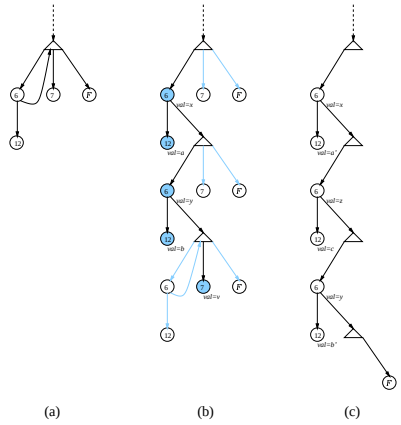

Figure 5. Detecting changes to repeatable nodes

Given s^i, d_j^k, d_j^l ($k < l$), to detect changes to repeatable, optional nodes:

Set the marker to be at the first node of every set of repeatable nodes in $s^i_{d_j^k}$.
Traverse $s^i_{d_j^k}$ and parse d_j^l.
For each node n_m in d_j^l, compare each n_m child, n_{mc}, with the child, n_{sc}, of the corresponding n_s in $s^i_{d_j^k}$ and n_{sc} is the marked node.

If $n_{mc} = n_{sc}$, return $upd(n_{mc})$.

If $n_{mc} \neq n_{sc}$

Compare n_{mc} with the rest of repeatable nodes of n_s sequentially.

If a match is found, return $upd(n_{mc})$, $move(n_{mc})$, remove n_{mc}'s matching from $s^i_{d^k_j}$, and mark the repeatable node immediately following n_{mc}'s matching in $s^i_{d^k_j}$.

If a match is not found, append n_{mc} to b^l_j.

return $del(n_v)$ for each n_v remaining of n_s and return $ins(n_w)$ from each $n_w \in b^l_j$.

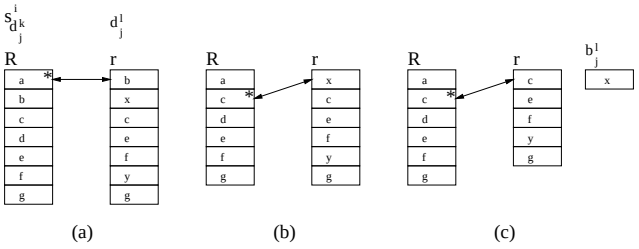

Figure 6. Comparing two sets of repeatable nodes

4.3. Cost

In this section, we discuss the complexity of the previous algorithms and the overall cost of detecting changes to data within a document. We consider the worst case scenario for upper-bound cost of our algorithm.

For the first three cases, the algorithm's cost is incurred during the parsing of the document. Thus, its order of complexity is linear in the number of objects in the document, as is the number of ordered nodes. It is linear since the maximum number of comparisons that must be performed is equal to the number of ordered nodes. In the case of unordered nodes, a given unordered node in d^l_j must first be compared with a given set of related unordered nodes in $s^i_{d^k_j}$. Since this is also performed during parsing of the document, the cost of detecting changes for unordered nodes is also linear. This observation also applies to the algorithm for nonrepeatable-optional nodes.

However, it is not the case for repeatable-optional nodes, where significant cost may be incurred. To better understand and evaluate the complexity of this case, we use two sets of linked lists consisting of the corresponding sets of repeatable nodes, where one is in $s^i_{d^k_j}$ and the other is in d^l_j. For the purpose of simplicity, let suppose that there was only one set of repeatable nodes in a document. In actuality, multiple sets of repeatable nodes is normal occurrence. An example is depicted by Figure 6.

In Figure 6(a), the left and the right lists represent the set of repeatable nodes in $s^i_{d^k_j}$ and the corresponding set of nodes in d^l_j, respectively. A box (e.g., "a") in the the list represents a repeatable node with or without its children (children may be necessary to help the comparison). The "*" represents the marker identifying which node in R the comparison should start with. Each node in r is compared with the list to its left starting with the marked node. Any pair of matching nodes is removed from both lists, e.g., node "b" of Figure 6(b). The marker is set to the following node "c" and the next node in r, "x", is compared with the left list starting from "c" and circled back to the beginning of the list, i.e., starting from "c" until "a". Since none of the remaining nodes in R matches "x", it is removed and stored in b^l_j as can be seen in Figure 6(c). The comparison process is repeated until all nodes in r have been evaluated.

In the worst case scenario, the complexity of the algorithm is $n \times m$, where n and m are the number of repeatable nodes in R and r, respectively. Let L denote the total number of ordered nodes, unordered nodes and nonrepeatable optional nodes in the new document. Let R and M denote the number of repeatable-optional nodes in the new document and the average number of repeatable nodes within one repeatable-optional node, respectively. Then, the total comparisons that need to be performed is $L + R \times M^2$. With R normally relatively small, in the worst case, the overall complexity is $O(N^2)$, where N is the number of nodes in the largest set of repeatable nodes in the document. Note that N is not the total number of nodes in the documents.

This can only occur when all of the corresponding repeatable nodes in the old version of the document have been deleted and replaced with a completely new set of nodes in the new document. For scientific documents, especially genomic documents, this scenario is very infrequent. Furthermore, relatively large portions of the new document usually maintain the same ordering as that of the older version. (Here, ordering refers to the location of a specified repeatable node with respect to two other repeatable nodes immediately preceding and following it). This is the reason for employing the marker in our algorithm. We are attempting, as much as possible, to exploit such characteristics so that the algorithm typically performs significantly better than $n \times m$. In practice, the algorithm is expected to perform $O(n + m)$.

5. Changes to schema

To detect changes to schema, we need the old schema graph and a document that conforms to the new schema. Our approach is based on the observation that some, if not all, of the schema changes will be reflected in future documents.

In particular, when the schema changes, some of the documents will fail to conform to the existing schema. As a result, changes that would normally be considered errors, such as an unrecognized identifier, are treated as schema modifications instead. In order to obtain the best reflection possible of the scope of schema changes, parsing should continue as much as possible after a schema change has been identified.

5.1. Types of schema changes

We define the output of a schema change to be a tuple $\langle st, sn, sp, Sib, CO \rangle$, where st, sn, sp, and v are the type of schema change, corresponding node identifier in schema graph, and parent node of sn respectively. Sib is a set of nodes that uniquely identify sn, and CO is the ordering of sn child-nodes from left to right as reflected in the document. In general, Sib and CO are optional.

Before we describe our approach to detecting schema changes, we present the different categories we consider, and the output for each operation:

1. **Reordering a node**. For a given node n_i, if the ordering of its children is significant, a reorder indicates it has changed. For example, the ordering between *COMPND* and *SOURCE* may be switched. We use $reorderS()$ to denote this operation.
   Output for $reorderS()$ is $\langle schemareorder, n_i, p_{n_i}, CO \rangle$ and CO is required.

2. **Inserting a new node**. An insert is considered to have taken place when an unrecognized identifier, n_i, is found. We use $insS()$ to denote this operation.
   Output for $insS()$ is $\langle schemainsert, n_i, p_{n_i}, Sib \rangle$.

3. **Deleting a node**. For a regular node n_i, a deletion is indicated by a document not containing a required child node. However, this may also indicate that it has been converted into an optional node. These options can be differentiated by evaluating a set of documents. Similarly, the deletion of an optional node requires an analysis of multiple documents. Our approach to this analysis is outlined in Section 5.3. We use $delS()$ to denote a deletion of a node.
   Output for $delS()$ is $\langle schemadelete, n_i, p_{n_i}, Sib \rangle$.

4. **Renaming a node**. If an unrecognized identifier matches a previously existing child of the same parent n_i, we consider that to be a rename of the existing identifier. We use $renameS()$ to denote an update of a node identifier on the schema.
   Output for $renameS()$ is $\langle schemarename, n_i, p_{n_i}, Sib \rangle$.

5. **Adding/deleting a repeatable edge**. This schema change is only applicable to a regular node whose ancestor is an optional node. The addition of a repeatable edge can be detected by evaluating a document where a node which previously never appeared multiple times now does. The appearance of multiple nodes that were previously mutually exclusive may also indicate the addition of a repeatable edge. For example, given the schema depicted by Figure 2 the appearance of n_7, n_6, n_{12} in a new document would signal the insertion of a repeatable edge on n_7. Detecting the removal of a repeatable edge is a more difficult task, and is described in section 5.3. We use $repeatS()$ and $nonrepeatS()$ to denote an addition and a deletion of a repeatable edge respectively.
   Output of $repeatS()$ is $\langle schemarepeat, n_i, p_{n_i} \rangle$, where n_i is the node that was previously non-repeatable and has appeared multiple times in the document.

5.2. Detecting changes to schema

For a node in the schema graph and an object in the document to *exactly* match, their identifiers must be identical. Two nodes having different identifiers partially match if more than a specified percentage of their children match, based on a matching among the children in which order is not considered. The number of children required to match can be adjusted depending on the level of accuracy desired.

For the remainder of this section, we assume that the document we are parsing conforms to a newer version of the schema than the parser. We first present the general algorithm for detecting schema changes, then describe in detail the approach for each type of nodes. Section 5.3 discusses how we infer complex schema changes, taking into consideration the possibility of errors, by analyzing schema changes from a collection of incomplete document comparisons.

This algorithm traverses the schema graph identifying schema changes while parsing a document:

$S \leftarrow$ root node of the schema graph
$D \leftarrow$ the beginning or root object of the document
While $S \neq \emptyset$ **and** $D \neq \emptyset$ **do**
  $d \longleftarrow pop(D)$
  $s \longleftarrow pop(S)$
  if s is a regular node **then**
    detect schema changes between d's children and s's children
    for any matching d's children, n_d, with the corressponding children, n_s of s **do**
      $push(n_d, D)$ and $push(n_s, S)$
  if s is an optional node **then**
    detect schema changes between d's children n_d and s's children
    if n_d matches a child-node, n_s, of s **then**

$push(n_d, D)$ and $push(n_s, S)$
if n_d is a repeatable node **then**
$n_{dx} \leftarrow$ location in the document immediately following n_d
$push(n_{dx}, D)$ and $push(n_s, S)$

S and D are lists of nodes. $push(y, L)$ and $pop(L)$ have the standard FIFO queue semantics.

When detecting schema changes, this algorithm relies heavily on the original schema. The more rigid the schema (i.e. the fewer optional nodes), the more changes that can be detected. Thus while this approach will work for semi-structured documents in general, it is most useful when applied to documents that are well constrained - such as scientific documents. For example, in PDB documents, the beginning and ending of a major identifier, such as *SOURCE* is easily identified by its tag along the left column. Additionally, characters such as {:;.,} and "tab" can be used to identify the beginning/ending of a data object or grouping of data objects.

Children of a regular node
Given node n_k from schema graph s^i and the matching object n_l within a document d_m where both n_k and n_l are regular nodes, to detect schema changes to children of n_k:

Traverse the sub-graph rooted at n_k creating a list, C_k, of n_k's children in order from left-to-right.
Identify all children of n_l, C_l, by parsing d_m starting from n_l until an identifier that matches one of the n_k's siblings is encountered.
Sequentially compare C_k and C_l to detect changes.
For every $n_v \in C_k$ that strictly matches an identifier,$n_w \in C_l$
If order of n_k's children on s^i is significant, then $reorderS(n_w)$ is returned.
For every unmatched identifier in $n_w \in C_k$ and $n_v \in C_l$
If n_v partially matches n_w, $renameS(n_w)$ is returned.
For every unmatched n_v, $insS(n_v)$ is returned.
For every unmatched n_w, $delS(n_w)$ is returned.

Children of an optional node
Given an optional node, n_k, on schema graph s^i, and the matching object, n_l, within a document d_m, to detect schema changes to children of n_k:

Traverse the sub-graph rooted at n_k identifying all children of n_k, C_k.
Fetch the next identifier, n_v, immediately following n_l in d_m.
If n_v matches one of n_k's sibling nodes,
n_l's children is a stopping node and no schema change is detected.
If n_v matches one of n_k's children, n_w
Fetch the next identifier from d_m.
If it matches one of n_k's children, n_v is repeatable.
If n_w is non-repeatable in s^i and n_v is a repeatable, then return $repeatS(n_v)$.
If n_v does not match any of n_k's children, then return $insS(n_v)$.
If n_w is a repeatable node that has been changed into a non-repeatable node, the approach described in Section 5.3 is used.

5.3. Inferring schema changes

Because a document is only one instance of its schema, it may not accurately reflect all changes to that schema. Thus, multiple documents may need to be considered at once to identify the new schema. To detect changes spanning multiple documents, we envision applying data mining techniques to a large collection of comparison results. Mining a large collection of schema changes addresses these problems:

1. Determining when to delete a child of an optional node.
2. Identifying unintended errors.

The larger the number of documents considered, the greater our confidence in the resulting patterns. For example, assume that we have n documents that conform to schema s^{i+1}. By comparing these documents against the previous schema, s^i, we identify a set of schema changes for each document Ch where $Ch = \{ch_{d_1}, ch_{d_2}, ..., ch_{d_n}\}$. We can then use statistical analysis in the following ways:

1. If n_l is the child of an optional node being considered for deletion and if Ch indicates that data objects corresponding to n_l do not appear in any of the documents, we may deduce with a level of confidence proportional to n that n_l has been deleted.
2. An error may be reflected by an $insS(), delS()$ or $upd()$ on a node. In Ch, an unintended schema change is likely to take place in only one or two documents. Schema changes occurring only in a very small percentage of documents are usually errors and should be ignored. However, because this could ignore valid schema changes, we assume that the changes are relatively minor, and do not ignore infrequent but significant schema changes.

More detail discussion on our schema analysis and its complexity can be found in [2]. We believe mining schema changes is a feasible approach to identifying complex modifications. However, human intervention may be required to

resolve some conflicts. The main objective of this effort is to automatically detect as many schema changes as possible so manual reconstruction of the new schema from scratch is no longer necessary.

6. Implementation within DataFoundry

We are currently implementing our approach within the context of the DataFoundry [8] project at Lawrence Livermore National Laboratory (LLNL). DataFoundry is a data warehouse that integrates scientific data from several distributed, autonomous, heterogeneous information sources. DataFoundry provides LLNL scientists with a uniform and semantically consistent interface to a variety of data. Figure 7 provides a simplified view of the DataFoundry architecture. Wrappers extract scientific documents from the underlying information sources, such as PDB, parse them, and pass the data to the mediator. The mediator transforms the data into the appropriate format, and propagates it to the warehouse. Because DataFoundry is focused on scientific data in general, and genomic data in particular, it faces the problems of detecting data and schema changes addressed in this paper. Typically, genomic information sources provide their schema as a free-formatted document, requiring manual identification of the locations and types of schema changes made to a new revision. Once these changes are identified, the associated wrapper is manually updated to conform to the new schema. This has proven to be costly and time consuming.

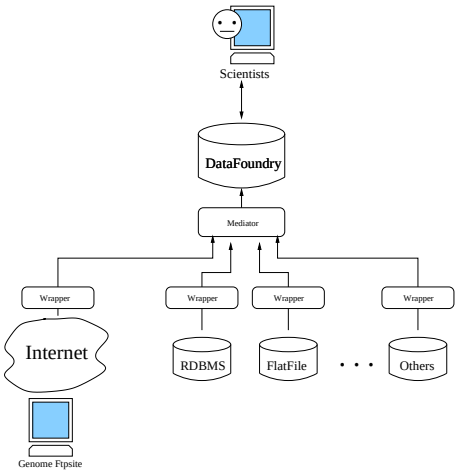

Figure 7. General Architecture for DataFoundry

We have developed a module that periodically detects and retrieves new documents from the information sources. These documents are then passed on to the appropriate wrappers to be parsed and entered into the warehouse. We are currently extending these programs to utilize the value-added schema to detect data and schema changes. At this time, data change detection has been implemented, and schema change detection is under development. We envision the change management architecture for DataFoundry, shown in Figure 8,which extends the current DataFoundry architecture by adding the *schema mining*, *schema storage* and *generator* components.

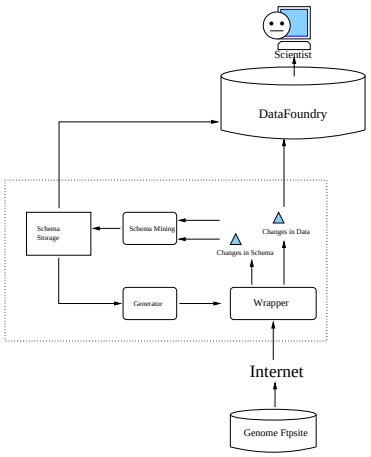

Figure 8. General Architecture for the change management system

In this architecture, when a change is detected, the wrapper retrieves all newly created or modified documents from the information source. Each extracted document is compared with its older version, stored locally by the management system. [1] Data changes are extracted and propagated into DataFoundry while schema changes are stored. By comparing several documents, and accumulating the schema changes, we obtain a better view of the new schema. The schema mining component then analyzes the collection of changes and defines a new schema. In doing so, human intervention may be required to resolve conflicts arising from incomplete or inconsistent data. The new schema is added to the schema storage, and is used by the generator to define a wrapper that conforms to it.

7. Conclusion

In this paper we have highlighted the importance of detecting changes to both data and schema, and proposed a for-

[1] In order to minimize storage requirement, only the newest version of each document is stored. Existing compression techniques can be used to further reduce the storage requirement

mal representation of semi-structured, scientific documents and their schema. We have addressed the problem of detecting data and schema changes by comparing the new document, with its implicit schema information, to the older version, represented as a value-added schema graph. We have presented the types of data and schema changes that may occur in scientific documents, and proposed detection algorithms accordingly. Our approach avoids performing extensive data matching between two versions of documents by performing change detection during the parsing of the documents, and by using the schema to guide the process. In order to minimize manual intervention for detecting schema changes and modifying existing wrappers, we have proposed a general purpose architecture and the necessary components for an automated change-management system. We are currently in the process of implementing this architecture within the context of the DataFoundry project at LLNL.

8. Acknowledgments

Adam and Adiwijaya would like to acknowledge the help and support from Lawrence Livermore National Laboratory.

A portion of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

References

[1] S. Abiteboul. Querying Semi-structured Data. In *Proceeding of ICDT*, January 1997.

[2] Igg Adiwijaya. Detecting data and schema changes in semi-structured scientific documents in data warehousing environment. In *Rutgers University Ph.D. Dissertation (Draft)*, 2000.

[3] Protein Data Bank. Protein Data Bank Contents Guide: Atomic Coordinate Entry Format. In *published at http://www.pdb.bnl.gov*, 1999.

[4] P. Buneman. Semistructured Data. In *ACM PODS*, 1997.

[5] S. Chawathe, S. Abiteboul, and J. Widom. Representing and Querying Changes in Semistructured Data. In *ICDE*, 1998.

[6] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogenous Information Sources. In *IPSJ Conference*, 1994.

[7] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *Proceedings of the ACM SIGMOD Conference*, June 1996.

[8] T. Critchlow, K. Fidelis, R. Musick M. Ganesh, and T. Slezak. DataFoundry: Information Management for Scientific Data. 4(1):52–57, March 2000.

[9] F. Douglis and T. Ball. Tracking and Viewing Changes on the Web. In *1996 USENIX Technical Conference*, 1996.

[10] F. Douglis, T. Ball, and Y. Chen. WebGUIDE: Querying and Navigating Changes in Web Repositories. In *Fifth International World Wide Web Conference*, May 1996.

[11] D.S. Hirschberg. Algorithms for the longest common subsequence problem. In *Journal of the ACM*, pages 664–675, October 1977.

[12] E. Myers. An O(ND) difference algorithm and its variations. In *Algorithmica*, volume 1, pages 251–266, 1986.

[13] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting Schema from Semistructured Dat. In *Proceedings of 1998 ACM International Conference On Management of Data (SIGMOD'98)*, June 1998.

[14] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative Objects: Concise Representations of Semistructured, Hierarchical Data. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, 1997.

[15] D. Sasha and K. Zhang. Fast algorithms for unit cost editing distance between trees. In *Journal of Algorithms*, volume 11, 1990.

[16] R. Wagner. On the complexity of the extended string-to-string correction problem. In *Seventh ACM Symposium on the Theory of Computation*, 1975.

[17] J. Wang, K. Zhang, K. Jeong, and D. Shasha. A System for Approximate Tree Matching. In *IEEE Transaction On Knowledge and Data Engineering*, volume 6, pages 559–570, August 1994.

[18] S. Wu, U. Manber, and E. Myers. An O(NP) sequence comparison algorithm. In *Information Processing Letters*, volume 35, pages 317–323, September 1990.

[19] K. Zhang, J. Wang, and D. Sasha. On the editing distance between undirected acyclic graphs. In *International Journal of Foundations of Computer Science*, 1995.